Nested Virtualization Without the Nest

Mathieu Bacou*
Grégoire Todeschi
IRIT, Univ. de Toulouse, CNRS
Toulouse, France
first.last@inp-toulouse.fr

Alain Tchana
I3S, Univ. Nice Sophia Antipolis, CNRS
Nice, France
alain.tchana@unice.fr

Daniel Hagimont
IRIT, Univ. de Toulouse, CNRS
Toulouse, France
daniel.hagimont@enseeiht.fr

ABSTRACT

With the increasing popularity of containers, managing them on top of virtual machines becomes a common practice, called nested virtualization. This paper presents *BrFusion* and *Hostlo*, two solutions that address each of two networking issues of nested virtualization: network virtualization duplication and virtual machine-bounded pod deployments. The first issue lengthens network packet paths while the second issue leads to resource fragmentation. For instance, in respect with the first issue, we measured a throughput degradation of about 68% and a latency increase of about 31% in comparison with a single networking layer. We prototype *BrFusion* and *Hostlo* in Linux KVM/QEMU, Docker and Kubernetes systems. The evaluation results show that *BrFusion* leads to the same performance as a single-layer virtualization deployment. Concerning *Hostlo*, the results show that more than 11% of cloud clients see their cloud utilization cost reduced by down to 40%.

KEYWORDS

virtualization, nested virtualization, container, network, orchestrator

1 INTRODUCTION

Virtualization is now well established as the base layer for cloud services. Virtual machines (VMs) present to the cloud user a customizable environment in the form of a complete operating system (OS). This customization is a burden to the user, who only wishes to quickly deploy his applications. Thus *containerization* — lightweight OS-level virtualization, such as Docker [25] — has emerged as the easier way to deploy and manage applications. First, building container images is easier than building VM images. Second, with containerization the host kernel has some knowledge about what is running inside a container, whereas VMs are seen as black boxes. This capability facilitates the configuration and deployment of complex applications, composed of several micro-services made of logically coupled containers, via a micro-service manager — an orchestrator, such as Kubernetes [16]. In the rest of the document we use the term *pod*, from Kubernetes's jargon, to refer to a micro-service.

Despite the popularity of the *Container-as-a-Service* (CaaS) model, we observe that the current practice is to deploy containers inside VMs, and thus VMs remain at a central position in cloud platforms. This virtualization stack is called *nested virtualization*, and it exists because of the two following main reasons.

Security. It is well known that containers are less secure than VMs [18, 19, 33] and offer worse resource isolation. On a cloud platform that hosts applications from different tenants, applications are isolated among each other using VMs while the components of one application are isolated using containers. This practice is very popular in the new cloud model of Function-as-a-Service [37] (FaaS) such as AWS Lambda: functions run inside containers, which in turn run in VMs [38].

Derivative clouds. The infrastructure of the cloud platform may be solely based on VMs, so that users are forced to create VMs before deploying containers. This practice is common in community clouds where the incentive to follow new trends is weaker, and administrative inertia weighs on technological updates. As an example, Atos Integration[1] is building a cloud platform for the European Space Agency (ESA) to exploit satellite imagery. This platform will be deployed on a third-party European VM-based IaaS cloud, while its usage is based on containers to facilitate the deployment and management of the clients' applications.

This paper identifies two issues in the current nested virtualization design.

Network virtualization duplication: host OS to VMs, and VM to containers. Both network virtualization layers use the same software components (e.g., bridge) to achieve virtual routing, thus *duplicating the network path*. We observed a throughput

*Also with Atos Integration.

https://doi.org/10.1145/3337821.3337840

[1] A world-wide IT services company.

degradation of 68% and a latency increase of 31% in comparison with a single networking layer. In this paper we present *BrFusion*, a nested network virtualization design which leads to the same performance as a single-layer virtualization system. To this end, *BrFusion* fuses the two virtualization layers by allowing containers inside a VM to directly use the host-layer networking stack.

VM-bounded deployments: a pod which is composed of several containers cannot be spread over several VMs, leading to resource fragmentation. This issue comes from the fact that container-level network virtualization is achieved per-VM, and specific technologies are required to seamlessly connect two containers across different VMs. These network overlays severely degrade inter-container communications [34, 42], while not actually achieving a transparent link between two containers of the same pod. Thus orchestrators only implement single-VM pod deployments [4]: containers belonging to the same pod must be deployed inside the same VM. This paper presents *HostLo*, a cross-VM nested virtualization design which allows cross-VM pod deployments.

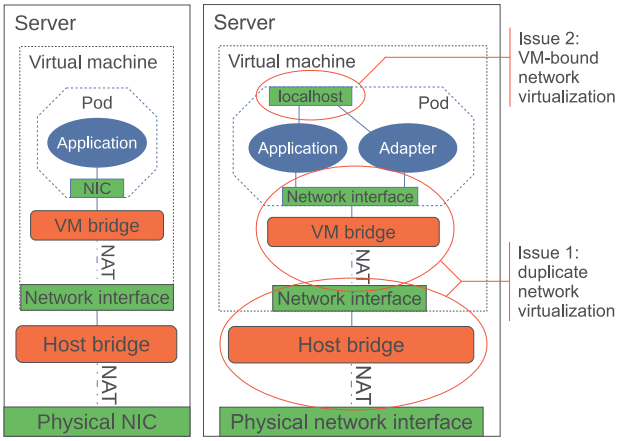
Solving both issues can be seen as *removing the nest from the nested virtualization*, from the client's point of view. We seek to make the pod orchestrator the main actor of the datacenter, by allowing it to communicate its orders to the virtual machine manager (VMM).

In summary, we make the following contributions:

- an analysis of nested virtualization effects on networking performance and resource usage;
- *BrFusion*, a de-duplicated networking stack that fuses both levels of network virtualization, improving network performance;
- *HostLo*, a network virtualization solution to enable cross-VM pod deployments;
- an implementation of *BrFusion* and *HostLo* in popular systems (Linux, KVM, Docker and Kubernetes).
- a thorough evaluation of our prototype in terms of performance, resource/money saving and overhead.

The evaluation results show that *BrFusion* leads to almost the same performance as a deployment with a single virtualization layer. In addition, it reduces CPU consumption inside the VM, thus improving the performance of concurrent CPU intensive containers. Finally, *BrFusion* incurs no overhead. About *HostLo*, simulations with Google cluster traces [29] show that it reduces the total number of VMs bought by a cloud user. We measured up to 40% of money saving per user. Concerning its overhead, *HostLo* reduces by 6.1 times and increases by 2.1 times request throughput and latency respectively, in worst cases. We show however that realistic scenarios do not suffer from this overhead. We compared *HostLo* with Docker Overlay [13]. *Hostlo* performs better than Docker Overlay, with up to 30% higher throughput and 92% lower latency.

The rest of the paper is structured as follows. Section 2 presents the motivations. Sections 3 and 4 present the contributions. Section 5 presents the evaluation results, as well as a discussion about our contributions. Section 6 presents the related work. Section 7 concludes the paper.



(a) Nested networking. (b) Pod networking issues for the adapter pattern.

Figure 1: Nested network virtualization: illustration and issues location.

2 NESTED VIRTUALIZATION: PROBLEM STATEMENT AND ASSESSMENT

In this paper we are interested in stacking containers inside VMs, which is the most popular nested virtualization practice. Although nested virtualization brings a lot of advantages (see section 1), its current design includes several issues that are discussed in this section.

The network, as a resource that can be virtualized, is very different from other resources such as CPU or memory. The difference comes from the fact that the network is actually two resources itself: (1) networking software and hardware such as protocol stacks, packet queues and physical ports; and (2) network identity, most commonly Ethernet MAC and IP addresses. Mutualizing the networking software and hardware facilities requires solving similar problems as mutualizing CPU or memory: how to arbitrate usage, how to isolate resources owned by each entity, etc.

However mutualizing a network identity is a very special case, which requires special operations that are specific to the technicalities of the network. For instance, in order to provide networking to its containers, Docker relies first on a bridge to mutualize the Ethernet MAC address of the physical NIC; and then Docker uses Network Address Translation (NAT) in order to forward packets to and from its bridge and the host's interface at the IP level. The setup with a bridge and NAT is the most common setup when using Docker as well as deploying VMs. This setup comes with two main issues for nested virtualization:

(1) duplicate network virtualization;
(2) VM-local network virtualization.

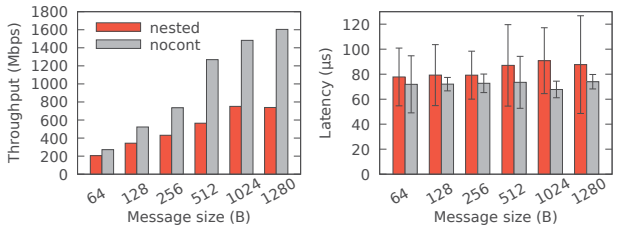


Figure 2: Network performance under nested and single-level (no container) virtualization. Excerpt from fig. 4.

These issues target different parts of a pod's network, as illustrated in fig. 1b. However, addressing them participates in the same global objective of *abstracting the VM layer from the orchestrator's point of view* — and thus from the client's point of view. We develop on these issues below.

Duplicate network virtualization. The network is virtualized twice: first at the host level, for the VMs; and second at the VM level, for the containers. This is illustrated in fig. 1a. This doubles the length of the packet reception and emission paths between the container and outside the virtual machine. Furthermore, both layers of network virtualization are managed by separate software solutions that *do not communicate* (respectively the VMM and the pod orchestrator), preventing any optimization of the data path.

A packet sent from the containerized application takes the following route (a received packet takes the same path in reverse): (1) the packet is placed on the pod's internal interface and crosses the pod's boundary; (2) it is received on the bridge in the VM; its destination is not any other interface on the bridge so it is not switched, but rather managed by NAT rules in the next step; (3) it is routed following NAT rules set up by the orchestrator using for instance iptables, and reaches the VM's NIC; (4) the packet reaches the host's bridge; (5) it is routed following NAT rules set up by the VMM, or the packet is switched by the bridge, to the physical NIC.

Figure 2 shows the impact of this design on the micro-benchmark Netperf. The server side runs in the nested virtualization context (a container in a VM) while Netperf client runs directly on the host physical machine, listening on a virtual interface NAT-ed to the host's bridge. We can observe a throughput degradation of 68% and a latency increase of 31% with 1280B messages compared to single-level virtualization (where Netperf server runs directly inside the VM). The solution we describe in this paper aims for the performance goal of single-level virtualization in the context of nested virtualization.

Constraint of VM boundary on network virtualization. At the VM level, the network is virtualized independently in each VM. It becomes a problem when we consider pod deployments, as all the containers in a pod must communicate with each other via a pod-private virtual network interface provided by Kubernetes, which by nature is local to the node that hosts the pod, i.e. the VM. *VM-local networking prevents cross-VM pod deployments*, which is a major loss for overall datacenter resource usage as well as for costs for the users, because it increases resource fragmentation. For instance, with Amazon AWS VM sizes [1], if your pod needs 6 vCPUs and 24GiB of memory, you must use a m5.2xlarge instance for $0.448/h because the entire pod must fit in the VM; however a m5.large and a m5.xlarge instances total up for 6 vCPUs and 24GiB for $0.336/h total. Thus enabling cross-VM pod deployments can save significant money and resources. This paper strives to achieve cross-VM pod deployment with acceptable degradation to the performance.

3 BRFUSION: NETWORK VIRTUALIZATION DE-DUPLICATION

In this section, we present *BrFusion*, a solution to the problem of network virtualization duplication in a nested virtualized environment.

3.1 Overview

Given that the host's NIC is already multiplexed between VMs via the host-level bridge (see fig. 1), we propose that the existence of a bridge in a VM comes from the false need of multiplexing the VM's NIC. Our solution revolves around the principle of *giving each pod its own NIC*.

Upon spawning the pod, a new NIC is provisioned by the VMM for the target VM. This interface is exclusive to the pod, so it can be directly inserted into the pod's network namespace, without the intermediary of NAT, a bridge and another vNIC in the VM to cross the namespace boundary. The new NIC is managed by the VMM that plugs it into a bridge on the host. Depending on the policy, it can be the common bridge used by all VMs on the host, or a tenant-specific bridge. In any case, the configuration is exactly the same as the current situation — i.e. it includes NAT, at the host level.

Given that the orchestrator is already a datacenter-global entity with local agents running inside each VM, the interaction between it and the VMM is very simple:

(1) the orchestrator asks the VMM for a new NIC to be added to the VM chosen during the scheduling phase, and optionally specifies the host-level networking domain (i.e. the bridge) that owns the new NIC;
(2) the VMM adds the new NIC to the VM and configures it accordingly;
(3) the VMM sends the orchestrator some sort of identifier of the new NIC so that its VM agent can use it (such as the MAC address);
(4) the orchestrator, via its VM agent, configures the NIC inside the VM and uses it for the scheduled pod.

3.2 Implementation

We prototyped *BrFusion* in QEMU/KVM environment. When QEMU creates a VM, it also provides a side-channel management interface [11]. It is easy to have it create a UNIX

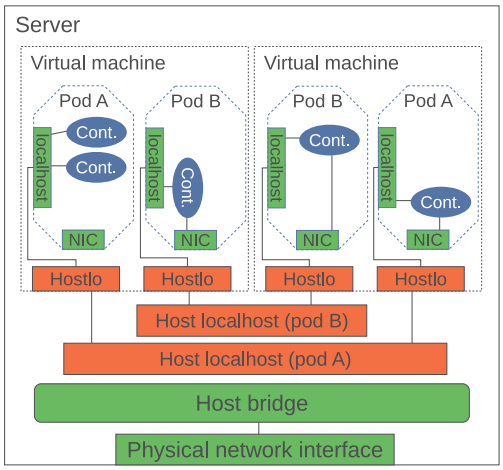


Figure 3: Host localhost: host-backed, cross-VM localhost interface for cross-VM pod scheduling. Hostlo interface B being above Hostlo interface A carries no meaning.

socket to which the VMM can connect. One of the many management actions the VMM can execute, is to add or remove NICs to and from the VM. The behavior is exactly the same as if the interface was provisioned at the start of the VM, and any modern OS is capable of detecting and using such hot-plugged devices.

Extending the Kubernetes orchestrator to ask the VMM for a new NIC when scheduling a pod is easily done with a Container Network Interface [3] plugin. CNI plugins follow a standard specification and are used to provide new networking models to Kubernetes.

4 HOSTLO: CROSS-VM POD DEPLOYMENT

In this section, we present *Hostlo* to enable cross-VM pod deployment in a nested virtualized environment.

4.1 Overview

Enabling efficient cross-VM intra-pod communication will remove the constraint of whole-pod allocation.

Intra-pod communication is achieved via *a localhost interface reserved to the pod*. This interface is provided by the VM that hosts the pod, which is why cross-VM pod deployments is currently impossible. The localhost interface is a virtual loopback networking device: simply put, it sends back any packet it receives, just as if the same cable was plugged both in the egress port and in the ingress port. Although an IP address is used to send and to receive packets on it, it functions at the link layer, manipulating Ethernet frames.

Our solution, shown in fig. 3, is to create on the host, a special loopback interface that can be multiplexed between several VMs. In each VM, an endpoint of this interface is used exclusively by the fraction of the pod that is placed there, as its localhost interface. This localhost interface is used by pods in VMs but backed by the host, thus the name *host localhost* (Hostlo).

In our solution, upon spawning the pod, if the placement decision involves more than one VM — the capability that we bring — the orchestrator needs to interact with the VMM:

(1) the orchestrator asks the VMM for a new Hostlo for the pod, and tells it to add that new interface to the VMs targeted for pod deployment;
(2) the VMM creates the new Hostlo, and multiplexes it between the specified VMs (i.e. inserts it as a new NIC in the VMs);
(3) the VMM sends the orchestrator some sort of identifier of the new NICs in the VMs so that its VM agent can use it (e.g. the MAC address);
(4) the orchestrator, via its VM agent, configures the NIC (i.e. the Hostlo endpoint) inside the VM and uses it for the scheduled pod.

4.2 Implementation

We prototyped *Hostlo* in QEMU/KVM environment. The orchestrator has no impact, because only the VMM does the hard work (provisioning a new Hostlo interface), although some integration similar to *BrFusion* is needed (see the end of the section).

The description of intra-pod communication given above guided the conception of our solution: *a virtual interface that sends back any packets it receives*, at the Ethernet frame level. This calls for the usage of TAP interfaces [12], that are exactly this: virtual network interfaces, provided by the Linux kernel, that accept Ethernet frames like a normal network device on one end, and that read and write Ethernet frames from and to a file descriptor. TAP devices are already used by QEMU as a backend for virtualized network devices.

Hostlo uses a TAP device, modified to act as a loopback interface, and that can be multiplexed among multiple VMs (which is not originally possible). We implemented[2] a modified TAP device driver in the host kernel so that:

- it provides at least one RX/TX queue for each VM that is served;
- it sends back any received Ethernet frame to all of its queues.

When QEMU is asked to provision a new *Hostlo* interface, it creates a new TAP device in loopback mode, and creates and add one RX/TX queue of it to each VM that needs it.[3] Creating and managing the TAP device is easily done by calling `ioctl`s on a *devfs* interface provided by the host kernel. Adding the *Hostlo* endpoints to the VMs is achieved in the same way as for *BrFusion* (see section 3), via QEMU's side-channel management interface.

Similarly to *BrFusion* in section 3.2, Kubernetes is extended with a CNI plugin in order to communicate with the VMM to create a new *Hostlo* interface that is used as the localhost interface of the new pod.

[2]Linux hostlo implementation: https://git.bacou.me/linux-vtap.
[3]QEMU hostlo implementation: https://git.bacou.me/qemu-hostlo.

4.3 Integration with other resources

Beside networking, containers in a pod also *share volumes* and can *communicate by shared memory* [4]. This means that enabling cross-VM pod deployment requires more than an efficient cross-VM localhost interface like *Hostlo*. However the issues of shared volumes and of shared memory are already addressed by previous works, as argued below.

4.3.1 Volumes. A volume, which is a limited section of the host's file system, can be mounted into a pod and shared among all its containers. It is used for data persistence beyond the life duration of the pod. All containers expect to be able to mount the volumes of the pod into their own file system.

Ultimately, reading from and writing to the volume is handled by the OS of the node where the pod is deployed. However in our context of disaggregated pod, more than one OS are involved. The OSes of the nodes do not expect to share the volume's file system with another OS, which would create inconsistencies due to in-memory file system states and guest caches. Thus solving the problem of cross-VM pod deployment from the point of view of volumes is not as simple as mounting the same file system in each VM that hosts a part of the pod.

Jujjuri et al. [20] designed a para-virtualized file system in QEMU/KVM called VirtFS. Based on VirtIO, it allows among other things, to mount the same file system into multiple guests. It is then a simple matter of synchronizing the orchestrator and the VMM to adequately mount the VirtFS into the VMs, and then the virtual volume into the parts of the pod.

4.3.2 Shared memory. For efficient intra-pod communication, the developer of a pod might want to use shared memory, where different containers share the same region of the node's memory to exchange data following a specific protocol. Memory sharing has been an important issue since VMs were used [30]. As long as it is solved at the VM level in a manner that is transparent to the processes, it is also solved at the pod level because memory management is not fundamentally changed by using pods.[4] Ren et al. [30] established an extensive survey of all techniques for cross-VM memory sharing; the best-suited solution for our context is MemPipe [41], which provides cross-VM shared memory on KVM at the transport layer, i.e. in a manner that is transparent to the containerized applications.

5 EVALUATIONS

This section presents the evaluation results of *BrFusion* and *Hostlo*. For each solution, we evaluate the gain it provides as well as the potential overhead that the solution could incur. Recall that we are interested in both application performance and resource/money saving. We measure the former on real hardware while the latter is simulated on real data center traces.

[4]Per-process virtual memory has been a fundamental concept to all modern OSes since before the concepts of containers or pods.

5.1 Setup

Environment. The machine used for real experiments is a Dell server with the following characteristics: 12 CPUs made available by two Intel® Xeon® E5-2420 v2 running at 2.20GHz (fixed under the PERFORMANCE governor) with HyperThread disabled. All VMs are provisioned with 5 vCPUs and 4GB of memory using QEMU with KVM, and run Arch Linux, kernel 4.19.9. Docker CE 18.09.0 is the container engine. All network interfaces in the VMs are based on virtio[31], and use Vhost[39] in their backend.

Benchmarks. To evaluate the performance gain and the overhead of our solutions, we use both micro- and macro-benchmarks. The micro-benchmark is Netperf [9]. We use Netperf's `UDP_RR` and `TCP_STREAM` benchmarking modes for latency and throughput evaluations respectively. `UDP_RR` measures request/response time by sending synchronous transactions, one at a time; while `TCP_STREAM` sends as much data as possible for a specified duration (20s). We measure the performance over different message sizes. The performance metrics, generated by Netperf client, are average request latency and average throughput. Concerning macro-benchmarks, we use three that are based on popular applications: Memcached, a key-value store; NGINX, a web server; and Kafka, a data streaming framework. Table 1 details the performance metrics and how the loads are generated.

Methodology. To evaluate the gain of *BrFusion*, we compare it with two solutions: (1) NAT: the default nested network virtualization solution; and (2) NoCont: no nested virtualization, which means that the application runs natively inside the VM, with no containerization. NoCont is the baseline, and represents the performance target of *BrFusion*. For each solution, we place the benchmark server in a VM, and the client runs on different CPUs of the physical host. It is linked to the host's bridge and to the VM via NAT. Beside network performance, we also evaluate how using *BrFusion* affects the VM and application CPU usage. We show the breakdown of CPU usage between: software work (noted usr); kernel work (noted sys), excluding interrupts handling; kernel serving software interrupts (noted soft); and host's CPU time given to a guest VM (noted guest). Finally, we evaluate the potential impact of *BrFusion* by comparing the boot time of containers with the vanilla nested virtualization solution (NAT).

About *Hostlo*, we evaluate its gain by comparing the cost of VMs used to host the pods of clients using Kubernetes (which only schedules whole pods on VMs) with *Hostlo* (with its capability to schedule a pod across multiple VMs). The simulation methodology is detailed in section 5.3. Concerning *Hostlo* overhead on application network performance, we compare it with: (1) NAT: see above; (2) Overlay: Docker's network overlay solution, which is the only currently viable approach for cross-node pod deployment; and (3) SameNode: all the containers of a pod run inside the same VM and they communicate via the pod's localhost interface. SameNode

Table 1: Macro-benchmarks: parameters and metrics.

Application	Benchmark	Parameters	Metrics
Memcached [6]	memtier_benchmark [7]	4 threads, 50 con./threads, SET:GET=1:10	Responses/s, latency
NGINX [10]	wrk2 [14]	2 threads, 100 con. total, 10k req./s on 1kB file	Latency
Kafka [2]	kafka-producer-perf-test.sh	120000 msg/s, 100B messages, batch size 8192B	Latency

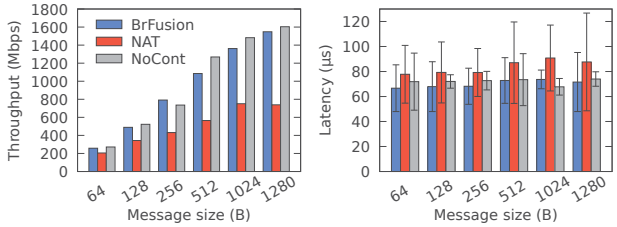
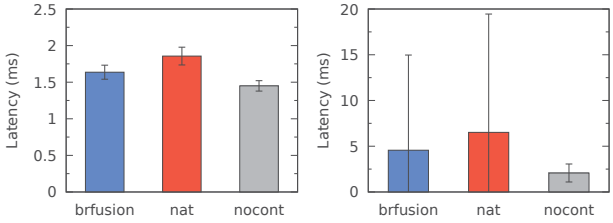


Figure 4: *BrFusion* performance gain using micro-benchmark. Bars are standard deviation of latency during the run.



(a) Kafka (b) NGINX

Figure 5: Kafka and NGINX results: latency. Bars are standard deviation of average latency across ten runs.

is the comparison baseline. We also evaluate the potential overhead on CPU usage.

The next two sections show the evaluation results of each solution. We begin by evaluating the gain, and then we check the potential overheads. Recall that *BrFusion*'s gain is mainly network performance, with a positive impact on CPU usage, while its overhead could be visible on container boot time. As for *Hostlo*, it aims at saving resources/money while its potential overhead could affect application performance.

5.2 BrFusion

5.2.1 Performance gain: micro-benchmark. Figure 4 presents the performance evaluation results. For both throughput and latency, *BrFusion* shows performance similar to NoCont, effectively removing the overhead of nested virtualization on networking. For instance, with 1280B packets *BrFusion*'s throughput is 2.1 times greater than NAT's and the average latency is 18.4% lower. *BrFusion* is also within 3.5% of NoCont's performance. Finally, *BrFusion*'s scales like NoCont with message sizes, while NAT scales more slowly and even stagnates between 1024B and 1028B.

5.2.2 Performance gain: macro-benchmarks. Evaluation setups are the same as above, with Netperf server replaced with the application and Netperf client replaced with the benchmarking tool. Figure 5 presents the results.

First for Kafka, *BrFusion* improves average request latency by 11.8% over NAT, which is 13.1% higher than NoCont. It also reduces the standard deviation, which is 5.9% of the average latency for *BrFusion* and 6.6% for NAT, compared to 4.9% for NoCont.

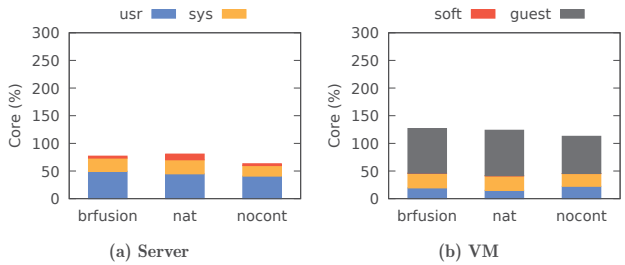
Second for NGINX, *BrFusion* improves on average request latency by 30.1% over NAT, but this is 120.3% slower than NoCont. Note that for both *BrFusion* and NAT, the standard deviation for the latency is about two times the average latency, while it is only 47% of the average for NoCont. Thus we attribute the main part of the performance overhead to the software itself rather than to the networking layer. *BrFusion* only removes the overhead of the latter.

5.2.3 CPU gain. By removing a layer of network virtualization inside the VM, *BrFusion* reduces the amount of CPU time used by the VM to perform network operations on behalf of containers it hosts. This results in CPU time freed for other applications on the VM. Thus we evaluated the amount of CPU time saved by *BrFusion* inside the VM.

Figures 6 and 7 present the measured breakdown of CPU usage for macro-benchmarks executions. For Kafka, *BrFusion*'s and NAT's CPU usages for the VM are both about 9.6% higher than NoCont's (fig. 6b). However the CPU usage of Kafka inside the VM shows that *BrFusion* reduces the CPU time spent serving software interrupts by 67.0% compared NAT (total reduction of 4.7%). Indeed, NAT rules[5] are applied on packets via hooks executed by software interrupts, and *BrFusion* simply removes the execution of these hooks. Similar observations of higher magnitude can be done for NGINX in fig. 7.

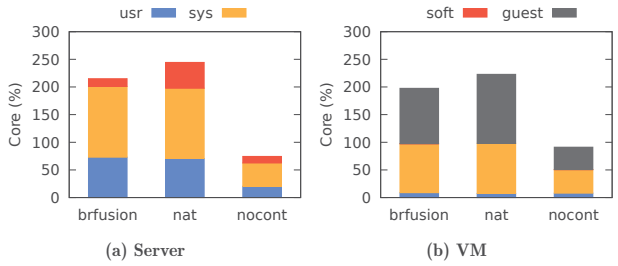
5.2.4 Overhead: container boot time. *BrFusion* provisions a new virtual NIC when creating a container (see section 3). This could harm the start up time of the container. We define the container start up time as the duration between ordering Docker to create the container, and the container sending a message through a TCP socket. We based our measurements on the Time Stamp Counter (TSC) [36]: we hacked QEMU to pass the physical TSC to the VM without any change, so the TSC acts as an absolute clock across the virtual boundary. This experiment was run 100 times. Start up times of containers with Docker NAT and *BrFusion*, are reported in fig. 8. Figure 8a shows that 75% of the measured start up times are slightly better with *BrFusion* than with Docker NAT. Figure 8b presents statistical figures;

[5]Implemented via Netfilter [8] in the Linux kernel.



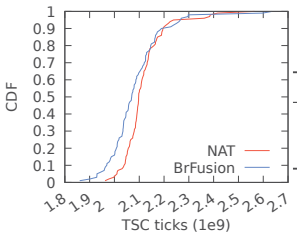
(a) Server (b) VM

Figure 6: Kafka results: CPU usage breakdown of the VM, and of the Kafka broker inside the VM. `guest` CPU time is time dedicated to the vCPUs of the VM.



(a) Server (b) VM

Figure 7: NGINX results: CPU usage breakdown of the VM, and of the NGINX server inside the VM. `guest` CPU time is time dedicated to the vCPUs of the VM.



(a) Cumulative distribution

	NAT	*BrFusion*	Diff.
Min	1.96	1.86	-5.2%
50%	2.10	2.07	-1.4%
95%	2.24	2.27	+1.3%

(b) Statistics (10^9 TSC ticks)

Figure 8: Container start up time evaluation: comparison of *BrFusion* with Docker NAT.

for instance, minimum start up time is about 5.2% better with *BrFusion*. The bottom line is that *BrFusion* does not slow down container creation time, and thus has no overhead.

5.3 Hostlo

We evaluate *Hostlo*'s gain in cost savings by simulation,[6] and its performance and CPU overheads on a real machine.

5.3.1 Cost saving. Recall that *Hostlo* enables spreading a pod's containers across several VMs on the same physical machine, thus reducing resource fragmentation. To evaluate the savings, we use Google Traces [29] to simulate per-user how much it costs to host the user's pods on VMs. Cost savings come from requiring fewer and smaller VMs to host a user's pod, thus it equates to resource savings. VM specifications and prices are taken from Amazon AWS EC2's on-demand `m5` models [1], and are reproduced in table 2. Resource specifications are values relative to the resources of the biggest model of VMs, `24xlarge`, similarly to resources given in Google traces.

Table 2: AWS EC2 VM `m5` models used to simulate *Hostlo* money savings.

Model	vCPU	Memory	vCPU (rel.)	Memory (rel.)	Price
large	2	8 GiB	0.0208	0.0208	$0.112/h
xlarge	4	16 GiB	0.0417	0.0417	$0.224/h
2xlarge	8	32 GiB	0.0833	0.0833	$0.448/h
4xlarge	16	64 GiB	0.1667	0.1667	$0.896/h
12xlarge	48	192 GiB	0.5	0.5	$2.688/h
24xlarge	96	384 GiB	1	1	$5.376/h

We compare *Hostlo* with Kubernetes using these steps:

(1) for each user, we begin with no VM and no pod;
(2) a user's pods are scheduled offline, biggest first;[7]
(3) a first scheduling is calculated by Kubernetes's:
  (a) try to schedule the whole pod on the already bought VM that best fits (see below), otherwise
  (b) buy a new VM to host the whole pod, of the size that best fits (see below).
(4) for *Hostlo*, we improve this scheduling by moving *containers* to the VMs that have the most wasted resources, smallest containers first, in the hope of eliminating the waste and reducing the number of needed VMs or shrinking the sizes of VMs — thus reducing costs.

When trying to schedule a pod, the algorithm chooses the best fitting VM according to Kubernetes's "most requested" policy [5]: among the VMs that have enough resources to host the pod, the best one is the one that currently has the most requested resources. [8] Simply put, this is a grouping strategy. When buying a new VM to host the pod, the best VM is determined as the cheapest one that can host the pod.

Although the described algorithm is rather crude, it allows to estimate fairly the money savings brought by *Hostlo* when compared to a classic pod scheduling. The simulation demonstrates satisfactory results, shown in fig. 9.

It shows the frequency of relative cost savings among 492 users in the Google traces. *Hostlo* reduces costs for about 11.4% of the clients, among which 66.7% show a costs reduction of more than 5%. The maximum relative cost savings are about 40%; the maximum cost save is about 237$/h, which represents a 35% reduction.

Below we evaluate *Hostlo*'s overheads. Although *Hostlo* brings benefits in terms of cost saving, it incurs two types of overhead. First, by spreading containers of the same pod

[6] Simulation code: https://git.bacou.me/hostlo-sim.

[7] The "size" of a pod is defined as the sum of its CPU and memory request.

[8] The most requested VM is determined by the average of CPU and memory relative requests made by pods on the VM.

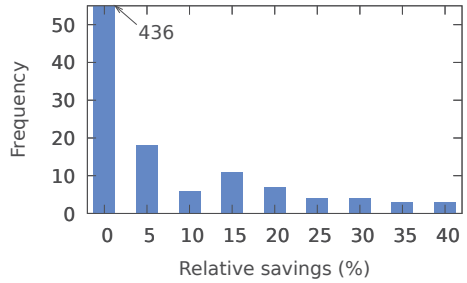


Figure 9: Money savings with Host localhost: frequency of per-user savings relative to Kubernetes.

across VMs, *Hostlo* lengthens the communication path between them. Recall that without *Hostlo*, a pod's containers communicate through the pod's localhost interface. Second, and for the same reason as the first, *Hostlo* increases the host's CPU usage. As it is implemented a kernel module of the host, this added load may be seen in the "sys" CPU usage category, i.e. improperly attributed to the host system rather than to guest VMs. We evaluate this overhead using both micro- and macro-benchmarks.

5.3.2 Overhead: micro-benchmark. Performance results are reported in fig. 10. When analyzing *Hostlo* performances, keep in mind that its purpose is container-to-container communication. This is typically comprised of small messages such as even notifications for logging and monitoring, and latency is of prime importance.

First about throughput, *Hostlo* scales with message sizes as well as NAT and Overlay, although the two latter show unexpected performance peaks for a few message sizes. As expected however, no solution reaches the performance level of SameNode. With a messages size of 1024B, *Hostlo*'s throughput is 17.9% higher than NAT's, 27% lower than Overlay's, and 5.3 times lower than SameNode's. Despite average throughput performance, *Hostlo* shows great latency results. Its latency remains stable across all message sizes, like SameNode; *Hostlo*'s latency is about twice SameNode's latency. As for NAT and Overlay, their latencies vary greatly and in unexpected manners. For a messages size of 1024B, *Hostlo*'s latency is 87.3% lower than Nat's, and 89.8% lower than Overlay's. The standard deviation of *Hostlo*'s latency is also rather low, about 27.9% of the average latency, while it is 20.5% for SameNode's latency. NAT and Overlay show standard deviations on latency between 25.8% and 95.4%.

5.3.3 Overhead: macro-benchmarks. We report performance results of Memcached and NGINX in figs. 11 to 13. For Memcached, *Hostlo* unexpectedly reaches the throughput and latency levels of SameNode. This is linked to SameNode showing extreme variability in its latencies in fig. 12. To the opposite, queries over *Hostlo* report stable latency. As for NGINX, *Hostlo* shows 49.4% higher latency than SameNode, but performs much better than NAT and Overlay. We observe very high standard deviation of the latency for all four

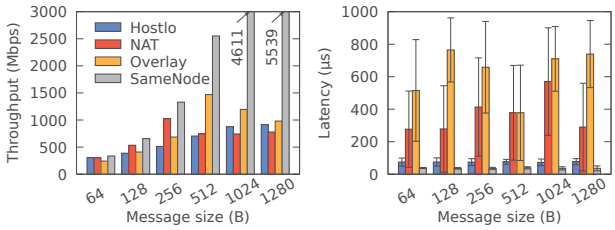


Figure 10: *Hostlo* overhead on network performance.

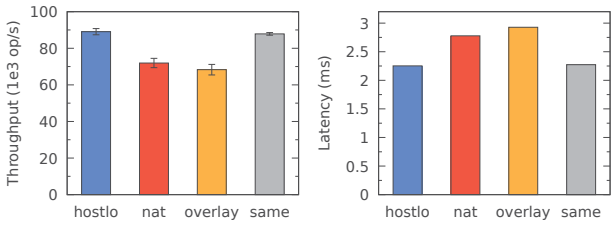


Figure 11: Memcached results: throughput and latency.

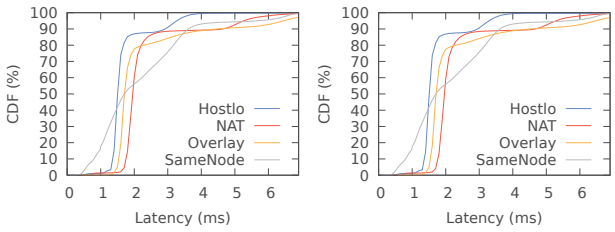


Figure 12: Memcached results: distribution of latency for GET and SET operations.

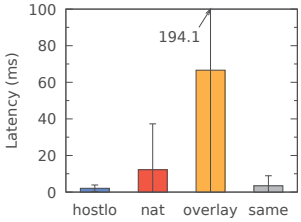
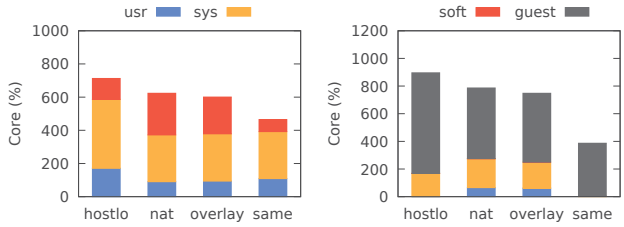


Figure 13: NGINX results: latency for 10k req/s on a 1kB file.

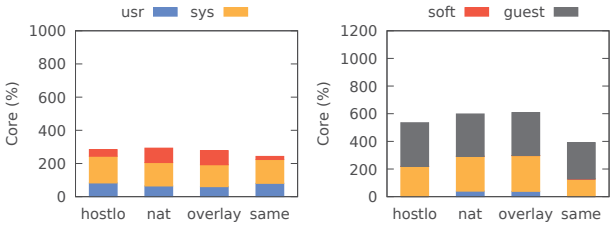
solutions. It is 3 times the average for *Hostlo*, and 1.6 times the average for SameNode.

5.3.4 Overhead: CPU usage. The extra CPU usage generated by *Hostlo* has been measured for macro-benchmarks (presented above). The results are shown in figs. 14 and 15. First for Memcached, compared to SameNode, the main increase due to *Hostlo* is the kernel CPU usage of the client



(a) Benchmark + server (b) VMs

Figure 14: Memcached results: CPU usage breakdown. Left: sum of CPU usage of the benchmark and the Memcached server in their respective VMs. Right: sum of the CPU usage of the VMs. For SameNode, there is only one VM. `guest` CPU time is time dedicated to the vCPUs of the VMs.



(a) Benchmark + server (b) VMs

Figure 15: NGINX results: CPU usage breakdown. Left: sum of CPU usage of the benchmark and the NGINX server in their respective VMs. Right: sum of the CPU usage of the VMs. For SameNode, there is only one VM. `guest` CPU time is time dedicated to the vCPUs of the VMs.

and the server, which is 46.7% more (fig. 14a). Total CPU usage of the client and the server increases by 53.2%. From the host, the CPU time given to the guests is increased by 89.8%, although it is important to note that by nature, the SameNode setup features only one VM, whereas *Hostlo*, NAT and Overlay include two VMs, which necessarily increases guest CPU usage. We can also observe that with *Hostlo*, some CPU time (1.68 core) is used by the host kernel on behalf of the VMs, notice however that around the same amount is also observed with NAT and Overlay. We conclude that this CPU time comes from the host kernel handling packets transiting through the guest's virtual interfaces and that are emulated with Vhost. Indeed, the latter is a host kernel component used by KVM to improve guest I/O by emulating it in the host kernel instead of letting QEMU manages a virtio device. It follows that the CPU usage of *Hostlo*'s host kernel module is correctly attributed to the guest VMs. Second for NGINX, the CPU increases of *Hostlo* compared to SameNode is much smaller: client and server CPU usage increases by 17.1%, and guest CPU usage increases by 36.9%. The same observations as for Memcached can otherwise be made.

Note that the CPU time consumed by the host on behalf of VMs due to *Hostlo* could be charged to the latter using [35], thus eliminating the overhead evaluated in this section.

6 RELATED WORK

Nested virtualization. Nested virtualization, as VMs-in-VMs, is used in derivative clouds [28], where the issue of network virtualization arose and overlay networks were developed. When containers, also known as OS-level virtualization, appeared, they replaced VMs at the nested layer. However, there has been work to bring VMs to the level of nimbleness of containers [23], that would resolve the issues that warrant using nested virtualization in the first place.

Virtualized networking optimization. Networking performance for containers has been investigated [17, 34]. In particular, Suo et al. [34] observed a drastic performance drop due to nested network virtualization.

Network virtualization is most often based on bridges [24], even with overlay networking, but they can be replaced by hardware switching using macvlan [15]. This is totally compatible with BrFusion, where the VM's bridge disappears and the interface inserted into the VM for the pod comes from the hardware switch created with macvlan on the physical host. However macvlan is not actually used by cloud providers because it requires them to change their core network routing, and to manage IP addresses allocation of containers [42]. Although better switching technologies can be used, such as macvlan or Open vSwitch [27], this does not solve the actual issues of duplicate network virtualization and VM-local networking for pods.

Container networking. Many other works improved container networking. Kim et al. [21] worked on virtual RDMA networking, while Zhuo et al. [42] developed kernel support for overlay networking. Overlay networking at the application level was also proposed by Subhraveti et al. [32]. These three solutions simplify and optimize overlay networking but they do not solve network virtualization duplication. Nakamura et al. [26] propose socket grafting, which bypasses container network stacks by grafting sockets in containers onto sockets in host network stacks, with proper access control. While it eliminates a network virtualization layer as BrFusion does, it requires to modify the applications or to trap syscalls. Also, all of these works do not take into account nested virtualization: they only optimize container-to-node communication, whereas BrFusion is about removing container-to-node communication at the transport level, so a container is directly linked to the virtualized network on the physical host.

Inter-VM communication. Optimizing communication between VMs has seen some work. One system has been proposed by Zhang et al. [40], combining shared-memory from Nahanni [22] for local VMs with SR-IOV over Infiniband for guests on different hosts, however it is only usable for MPI workloads. Like many other works, it requires to adapt the application, while Hostlo is a transport-level solution that requires no change to the containerized workload. Zhang and

Liu [41] developed MemPipe that works below the IP level to deliver packets to co-resident VMs via shared memory. While it does not require any modification to the applications, there is no concept of isolation: it simply analyzes packets to determine which VM is to be notified when transmitted data is ready. Leveraging this solution to transparently replace a pod's localhost interface would also be a challenge.

7 CONCLUSION

We proposed in this paper *BrFusion* and *Hostlo*, two solutions to two issues of nested virtualization: network virtualization duplication, and VM-bounded pod deployments. We prototype them in Docker, Linux KVM/QEMU and Kubernetes systems. The evaluation results show that our solutions outperforms state-of-the-art solutions with acceptable overhead.

We think that nested virtualization is useful in its own right, but some work it still needed to overcome its performance and management disadvantages. We believe that the way forward for nested virtualization is to clearly put the orchestrator as the only manager of the datacenter, and to integrate the VMM as a tool for the orchestrator.

REFERENCES

[1] 2019. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/. [Online; accessed March 2019].

[2] 2019. Apache Kafka. https://kafka.apache.org. [Online; accessed March 2019].

[3] 2019. Container Network Interface. https://github.com/containernetworking/cni. [Online; accessed March 2019].

[4] 2019. Kubernetes concepts: pods. https://kubernetes.io/docs/concepts/workloads/pods/pod/. [Online; accessed March 2019].

[5] 2019. Kubernetes source code. https://github.com/kubernetes/kubernetes/. [Online; access April 2019].

[6] 2019. Memcached. https://memcached.org. [Online; accessed March 2019].

[7] 2019. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark. [Online; accessed March 2019].

[8] 2019. The netfilter.org project. https://www.netfilter.org/. [Online; accessed April 2019].

[9] 2019. Netperf. https://github.com/HewlettPackard/netperf. [Online; accessed March 2019].

[10] 2019. Nginx. https://nginx.org. [Online; accessed March 2019].

[11] 2019. QEMU Machine Protocol. https://wiki.qemu.org/Documentation/QMP. [Online; accessed March 2019].

[12] 2019. Universal TUN/TAP device driver. https://elixir.bootlin.com/linux/v4.19.34/source/Documentation/networking/tuntap.txt. [Online; access April 2019].

[13] 2019. Use overlay networks. https://docs.docker.com/network/overlay/. [Online; accessed April 2019].

[14] 2019. wrk2. https://github.com/giltene/wrk2. [Online; accessed March 2019].

[15] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon. 2016. Performance considerations of network functions virtualization using containers. In *2016 International Conference on Computing, Networking and Communications (ICNC)*.

[16] Eric A Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of ACM Symposium on Cloud Computing*.

[17] Joris Claassen, Ralph Koning, and Paola Grosso. 2016. Linux containers networking: Performance and scalability of kernel modules. In *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP*. IEEE, 713–717.

[18] Theo Combe, Antony Martin, and Roberto Di Pietro. 2016. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing* 3, 5 (2016), 54–62.

[19] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang. 2017. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[20] Venkateswararao Jujjuri, Eric Van Hensbergen, Anthony Liguori, and Badari Pulavarty. 2010. Virtfs – a virtualization aware file system pass-through. In *Ottawa Linux Symposium (OLS)*.

[21] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *16th USENIX Symp. on Networked Systems Design and Implementation*.

[22] A Cameron Macdonell et al. 2011. *Shared-memory optimizations for virtual machines*. Uni. of Alberta Edmonton, Canada.

[23] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, 218–233.

[24] Victor Marmol, Rohit Jnagal, and Tim Hockin. 2015. Networking in containers and container clusters. *netdev 0.1* (2015).

[25] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* (2014).

[26] Ryo Nakamura, Yuji Sekiya, and Hajime Tazaki. 2018. Grafting sockets for fast container networking. In *Symposium on Architectures for Networking and Communications Systems*. ACM.

[27] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 117–130.

[28] K. Razavi, A. Ion, G. Tato, K. Jeong, R. Figueiredo, G. Pierre, and T. Kielmann. 2015. Kangaroo: A Tenant-Centric Software-Defined Cloud Infrastructure. In *2015 IEEE International Conference on Cloud Engineering*.

[29] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*. San Jose, CA, USA.

[30] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, Jianbo Guan, Jinzhu Kong, Huadong Dai, and Lisong Shao. 2016. Shared-memory optimizations for inter-virtual-machine communication. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 49.

[31] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems* (2008).

[32] Dinesh Subhraveti, Sri Goli, Serge Hallyn, Ravi Chamarthy, and Christos Kozyrakis. 2017. AppSwitch: Resolving the Application Identity Crisis. *arXiv preprint arXiv:1711.02294* (2017).

[33] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security namespace: making Linux security frameworks available to containers. In *27th USENIX Security Symposium (USENIX Security 18)*.

[34] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. 2018. An Analysis and Empirical Study of Container Networks. In *Proceedings of IEEE Conference on Computer Communications*.

[35] B. Teabe, A. Tchana, and D. Hagimont. 2016. Mitigating Performance Unpredictability in Heterogeneous Clouds. In *IEEE International Conference on Services Computing (SCC)*.

[36] G. Tian, Y. Tian, and C. Fidge. 2008. High-Precision Relative Clock Synchronization Using Time Stamp Counters. In *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*. 69–78.

[37] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup. 2018. Serverless is More: From PaaS to Present Cloud Computing. *IEEE Internet Computing* 22, 5 (Sep. 2018), 8–17.

[38] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference*.

[39] Wei Wang, J Nakajima, M Ergin, J Tsai, G Xiao, M Koujalagi, H Xie, and Y Liu. 2016. Design of Vhost-pci. In *KVM Forum*.

[40] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K Panda. 2017. Designing locality and NUMA aware MPI runtime for nested virtualization based HPC cloud with SR-IOV enabled InfiniBand. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 187–200.

[41] Qi Zhang and Ling Liu. 2016. Workload adaptive shared memory management for high performance network I/O in virtualized cloud. *IEEE Trans. Comput.* 65, 11 (2016), 3480–3494.

[42] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *16th USENIX Symposium on Networked Systems Design and Implementation*. 331–344.